AMIGA SHOPPER

Complete Amiga C

Special sample edition containing two complete chapters

Cliff Ramshaw

The shareware DICE C compiler is on the cover of the Amiga Shopper January '94 issue!

Future Leisure Books

Special preview sample FREE with the Jan '94 issue of Amiga Shopper

'Complete Amiga C' comes with compiler, libraries, includes, documentation and examples!

Complete Amiga C

Special sample edition containing two complete chapters

Cliff Ramshaw

Future LEISURE BOOKS

Book design and production Rod Lawton

Cover design Rod Lawton

Cover illustration Paul Kidby

Author Cliff Ramshaw

First published in 1993 by Future Publishing, Beauford Court, 30 Monmouth Street, Bath, Avon BA1 2BW

Printed in the UK by Garden House Publishing

Contents

Installing DICE

What follows is an explanation of how to install a working version of DICE, as supplied on the cover disk with *Amiga Shopper 33*. Please read and carry out the instructions carefully.

You'll need to open a Shell window and use some simple Shell commands. If you're unsure how to do this, consult your Amiga's manual.

The freeware version of DICE supplied will work with version 1.3 or above of the Amiga operating system. If you're still using 1.2, don't you think it's about time you upgraded?

The instructions you need to follow vary, depending on the operating system you are using, and whether you want to install to floppy disks or hard disk. Please follow the correct procedure below.

Installing DICE to floppies

You will need two blank formatted floppy disks. Format them using the Workbench. Consult your Amiga manuals if you are unsure how to do this. Rename these disks as **`DiceDisk1`** and **`DiceDisk2`**. It's important that you ensure that the disk names are correct for the rest of the installation to work.

In the drawer called **`DICE`** on the cover disk there are two files:

```
dice_part_1.lha
dice_part_2.lha
```

These are archive files. Although they only appear as two files, each one in fact contains many other files. The first file, **`dice_part_1.lha`** contains all the files that should go on the first floppy disk you formatted, **`DiceDisk1.`** The second contains the files for the second of the disks. Open a shell and type the following command in. At the end of each line, press the **`<Return>`** key.

```
AS_Vol_VII:c/lha x AS_Vol_VII:Dice/dice_part_1.lha DiceDisk1:
```

Now follow the instructions on screen, inserting disks when asked. If you have two drives, use them both to speed up this operation. When this is complete, and you get your Shell prompt back again, you need to perform a similar process for disk two:

```
AS_Vol_VII:c/llha x AS_Vol_VII:Dice/dice_part_2.lha DiceDisk2:
```

You now have two disks with the Freeware version of DICE set up on them. Neither of these disks are "boot disks", so you will have to boot from a Workbench disk whenever you want to use DICE. Make a copy of your Workbench disk, and call it something unique, such as **DiceBench**. You can make a copy either using the **diskcopy** command from the Shell, or by using Workbench. The **rename** command will enable you to give the disk the name **DiceBench**. This will be your DICE boot-disk.

If you are using Workbench 1.3

Boot off your new **DiceBench**. Open a Shell and type:

```
ed s:startup-sequence
```

This loads the editor. For proper instructions on using this editor, consult your Amiga manual. Go to the bottom of the loaded file (your startup-sequence), and *before* the **endcli** at the bottom, add the following four lines:

```
assign dice: DiceDisk1:
assign dlib: DiceDisk2:dlib
assign dinclude: DiceDisk1:include
setenv DCCOPTS "-1.3"
```

Now proceed to the section below entitled **Final Steps**.

If you are using Workbench 2.0 or higher

Boot from your new **DiceBench** disk. Open a Shell and type:

```
ed s:user-startup
```

This loads the editor. For proper instructions on using this editor, consult your Amiga manual. Add the following four lines to the bottom of this file (which may currently be empty):

```
assign dice: DiceDisk1:
assign dlib: DiceDisk2:dlib
assign dinclude: DiceDisk1:include
setenv DCCOPTS "-2.0"
```

Final Steps

Save the file and quit from the editor. Boot from **DiceBench** and then, using the Shell, type:

```
ed s:shell-startup
```

And add this line to the bottom:

```
path dice:bin add
```

Save the file and quit the editor, then type this final line into the Shell:

```
copy DiceDisk1:s/.edrc.floppy DiceBench:s/.edrc
```

You now have your DICE disks set up. If you have multiple disk drives, then you'll have less disk swapping to do, both during the setting up process and subsequently when you are using DICE to write programs. When you boot from your new **DiceBench** you will be asked to insert disks at various points during the boot process.

You can now skip to the section entitled **Using DICE**.

Installing DICE on a hard disk

Choose where you are going to place your version of DICE. Create a new drawer called **Dice** there. For example, if you wished to install DICE in your **Work:** partition, create a drawer there. You can do this from the Workbench if you have Workbench 2.0 or above, or by using the Shell if you have 1.3; consult your manual if you are unsure.

I will assume that you created a drawer called **`Dice`** in the **`Work:`** partition. If this is different, replace all occurances of **`Work:Dice`** in what follows with the appropriate path.

Oen a Shell and type in the following commands. At the end of each line, press **`<Return>`**. Each time you do so, you will see lots of information scroll up on the screen. This is the *lha* program extracting all of the individual files that have been compressed together into the two archive files.

```
AS_Vol_VII:c/lha x AS_Vol_VII:Dice/dice_part_1.lha work:Dice/
AS_Vol_VII:c/lha x AS_Vol_VII:Dice/dice_part_2.lha work:Dice/
```

If you are using Workbench 1.3

Using the Shell still, type this:

```
ed s:startup-sequence
```

This loads the editor. For proper instructions on using this editor, consult your Amiga manual. Go to the bottom of the file loaded in (your startup-sequence), and *before* the **`endcli`** at the bottom, add the following four lines:

```
assign dice: work:Dice
assign dlib: work:Dice/dlib
assign dinclude: work:Dice/include
setenv DCCOPTS "-1.3"
```

Now proceed to the section below entitled **Final Steps**.

If you are using Workbench 2.0 or above

Using the Shell, type the following:

```
ed s:user-startup
```

This loads the editor. For proper instructions on using this editor, consult your Amiga manual. Add the following four lines to the bottom of the loaded file:

```
assign dice: work:Dice
assign dlib: work:Dice/dlib
assign dinclude: work:Dice/include
setenv DCCOPTS "-2.0"
```

Final Steps

Save the file and quit from the editor. Using the Shell, type:

```
ed s:shell-startup
```

And add this line to the bottom:

```
path dice:bin add
```

Save the file and quit the editor, then type this final line into the Shell:

```
copy Dice:s/.edrc.hd s:.edrc
```

Now re-boot your Amiga to ensure that these changes you made take effect. DICE is now installed on your system.

Using DICE

DICE is now all set up and ready to run. If you're working from hard disk, then no problem; if you're working from floppy, then don't forget that you need to boot with your Workbench copy called **`DiceBench`** first. Because of a peculiarity of AmigaDOS, you'll find that you must have **`DiceDisk1`** in your drive when you use any of the DICE commands, such as **`dme`** and **`dcc`**, stored in the disk's **`bin`** directory. Otherwise, you'll get a "command not found" error message. This only applies to floppy disk-based systems.

There are four main stages to writing a program in C: editing, compiling, linking and running.

1. Editing

All DICE operations should be carried out from the Shell, so the first thing to do is to open one. Next, you need to decide where you are going to save your work. The RAM disk is a handy and speedy place, but of course you'll loose everything in the event of a crash or when you come to turn the power off. You might like to have a blank floppy disk ready instead, or to make a drawer on your hard disk.

All of the examples in this book are in a form known as "source code". You enter the programs using a standard ASCII text editor. The program *Ed*, which you used earlier to modify your startup-sequence, is one such program, but DICE comes supplied with a much better editor called *DME*. You can run *DME* from the Shell buy simply typing:

```
dme
```

Normally, you'll want to supply *DME* with the name of the file in which you want to store your source code. Let's assume you're going to be storing your code on a floppy disk called **`MySource:`** (you'll need to change the path in the line below if you intend storing your programs elsewhere). If your program is going to be called **`example.c`** (it's conventional to put a **`.c`** extension on the end of all C source code files), then you would type:

```
dme MySource:example.c
```

(Floppy users note that you must have **`DiceDisk1`** in a drive *before* typing this command.) This will open the editor with an empty file. Just type a program in (the first example in the following chapter is a good place to start), then select **`<Save-Quit>`**. You're now ready for the next stage.

2 & 3. Compiling and linking

Compilation is the process of translating source code written in C into machine code, a language executable by the Amiga's central processor. Compilation is the main function of DICE. After a program has been compiled, it needs to be linked. The process of linking joins the compiled program with any pre-written library programs that it might require to run. Fortunately, DICE also handles linking. Indeed, you can carry out both processes with a single command line. Assuming you've created the file **`example.c`** with the editor as mentioned above, then you can compile and link it with the following line :

```
dcc MySource:example.c -o MySource:example
```

(Floppy users note that you must have **DiceDisk1** in a drive *before* typing this command.) The actual command to make DICE compile and link is **dcc**. After this must come the path and name of the source file to be translated. The **-o** option enables you to specify the path and name of the output file, which makes up the final part of the command line above.

Notice that the paths used specify that the source code is to be found on the disk **MySource**, and that the compiler's output, the executable file called simply **example**, is to be put there too. You could put the resulting program elsewhere by specifying a different path; likewise you could give it a different name by using a different word instead of **example** at the end of the **dcc** command line.

If all goes well, you will be left with an executable file called **example**, residing on a disk called **MySource** (if you're using a floppy-based system you'll have to carry out a few disks swaps to get this far – the Amiga will ask for the disks as it needs them). If you've been given an error message, then you've probably made a typing mistake when entering your source code. Load up the editor again and check what you have entered. Once you have made your corrections, save your work and go back to the compilation stage.

4. Running your program

Once you've successfully compiled your program, all that remains is to run it. You can do this as you would for any ordinary AmigaDOS command, simply by typing its name (and including its path if it is not somewhere already in AmigaDOS's path). If you've saved the compiled program to your disk called **MySource**, then type:

```
MySource:example
```

And all being well, you should get the words "Hello from planet C" on the screen. Thats it!

There is a lot of documentation supplied with DICE on disk that will help you use the system in a way that's best for you. The documents are in the **`doc/`** drawer. For floppy users, this is **`DiceDisk2:doc/`**. For hard disk users, it's **`dice:doc/`**. Please do make the effort to read these files; they will help you solve most of your problems.

Registered and shareware DICE – the differences

The version of DICE supplied on the cover disk accompanying this book is freely-distributable. It is a fully-functioning compiler environment, but is lacking some of the extra features provided with the registered version of DICE. Registering DICE costs $50; you can find details on how to do so in the file called **`Register.doc`** in the **`doc/`** drawer of the DICE disks. Alternatively, if you buy the full edition of *Complete Amiga C* you will also get the fully-registered version of DICE as part of the package.

Advantages of the registered version include:

- Full set of Commodore include files amiga.lib (essential for tapping the full graphical and audio power of the Amiga).
- Extensive documentation and many example programs.
- Source code for the C library.
- Support for bitfields and **`_chip`** and **`_geta4`** keywords.
- Support for floating point maths.

This last feature needs some explaining. The shareware version of DICE will not handle floating variables (which is to say numbers with fractional parts). Some of the examples in this book require floating point support to compile, and you'll need to modify them to get them to run with the shareware version of DICE.

Here's how: Whenever you come across the keyword **`float`** in a piece of source code, change it to **`int`**. Likewise, whenever you see the two characters **`%f`** within quotes inside a **`printf`** or **`scanf`** statement, replace them with the new characters **`%d`**. If any values are assigned to what was previously a floating point variable, then you must remove the decimal point and anything after it from the value to be assigned.

These changes will convert the example programs to work with integer variables instead of floating point types. The reason for performing them will become apparent as you read through the next two chapters. They are only necessary if you are using the shareware version of DICE. The registered version, or a commercial compiler, will compile the unmodified programs without problems.

Commodore includes

The Commodore includes are a set of files that, along with a file known as amiga.lib, enable the C programmer to take full advantage of the Amiga's facilities. With the freeware version of DICE you can perform simple input and output via the Shell, but calling on the Amiga's built-in routines to open screens and windows, play samples and so on is not possible without Commodore's include files. If you're serious about programming the Amiga, you need these files. There are three ways you can get them. The first, and the easiest, is to buy 'Complete Amiga C' for £24.95, which comes with a fully-registered version of DICE and Commodore's include files. The second is to register DICE with its author, Matt Dillon, for $50. You'll find details on how to do this in the doc file `register.doc`. The third way is to send a cheque for £25, made payable to Commodore Business Machines (UK) Ltd, to Sharon McGuffie, Commodore Business Machines (UK), Commodore House, The Switchback, Gardener Road, Maidenhead, Berks. SL6 7XA, and ask for the Native Developer's Toolkit.

'Complete Amiga C'

If you like what you've seen in this sample, why not go ahead and buy the complete book? 'Complete Amiga C' costs £24.95 and includes the following:

- Complete beginners' guide to C programming
- Registered version of DICE
- All necessary Commodore includes
- On-disk DICE documentation

See Amiga Shopper for more information

Basic C programming

- Analysing a simple program
- Numbers and variables
- Variable types
- Expressions, operators and operands
- Handling user input
- Mixing variable types

All of what's gone so far may have given you the impression that programming is complicated. In fact, it's fairly straightforward, as you'll discover once you get those basic concepts under your belt. And the best way to do that is to start programming.

Analysing a simple program

The first thing to attempt is a program to print a message to the screen. There's hardly any problem-solving to be done at all: we already know that there's a C library holding a function to do the printing for us, so all we need to do is make use of that function. Everything else is just the basic framework which must surround all programs.

Run your editor and enter into it the following lines of code:

COMPLETE LISTING

```
#include <stdio.h>

/* program to print a message to the screen */
void main()
{
    printf("Hello from planet C\n");
}
```

Now save the text file, compile it and run it (you'll find instructions for using DICE to do this in Chapter Two).

You should see the text below printed on your screen:

```
Hello from planet C
```

I'll go through the program line by line to explain what's going on. The first line tells the compiler that we want to use some code that's already been written, collected in what is called a 'header file'. It tells the compiler to include this code as part of our own program. The name of the file to be included is held within the angled brackets. **Stdio.h** (for standard input and output) is a file provided with all C compilers, containing a set of functions to perform simple printing operations, retrieve entries from the keyboard and the like.

We need to include it because it contains the function to print information to the screen.

The next line has been left blank. It isn't necessary, but has been put there to aid clarity. So far is C is concerned, all white space is the same. It does not distinguish between a space character, a tab or a return character. They are all used merely as separators. It's possible, though confusing to read the result, to write C programs as a single line with only single spaces separating the statements.

Use comments in your code

The next line is a comment. It's a message from the programmer to him or herself, and to any other programmers who may look at the code. It's a signpost that explains what's going on. Using comments in your programs is a good idea: you'll be surprised how confusing your own programs can appear when you come to look at them some time after you first wrote them. Comments help you to unravel the knots. In C, the two characters **/*** together denote the beginning of a comment, and the characters ***/** denote its end. Anything between them is ignored by the compiler, and is not translated into machine code. In other words, comments don't waste space and don't detract from the efficiency of your programs. On the other hand, they do improve the efficiency with which you can root out program errors and make changes.

After the comment comes a function definition. The function is called **main**; every C program has a function with this name. It is this function that gets used when the program is executed. The reason such a function needs defining will become apparent if you consider that many more function definitions may be contained in a program; the computer must be told which one of them to execute first, which is the 'top level' or 'main' function. The word **void** before the function's name indicates that the function does not produce a result. It has an effect – that of printing text to the screen – but no result is produced in the strict sense that I keep promising to define in Chapter 6. The parentheses after the function name let the compiler know that it is a function that is being defined, and the fact that there is nothing between them indicates that the function does not require any input.

The opening and closing curly brace are used to bound the contents of the function. Everything between them, in this case just one statement, is defined as the block that comprises the function. They are just part of the grammatical structure that C expects, in much the same way as English requires reported speech to be enclosed by inverted commas.

Finally, we get to the statement that actually *does* something. What it does is to make use of (or "call" in technical parlance) a function named **`Printf`**. **`Printf`** is a pre-written function that writes information to the screen. Its definition is provided by the inclusion of the **`stdio.h`** header file. The "f" in its name stands for "formatted"; **`printf`** is a clever little function that can, if used properly, write all sorts of information to the screen in all sorts of different formats.

In this case our use for **`printf`** is straightforward. The information enclosed in the parenthesis is the input for the function (known as the function's "parameter"), for our purposes a segment of text. Enclosing items within quotation marks ensures that the compiler treats them as text.

The two characters at the end of the text – **`\n`** – are instructions that tell printf to print a newline character, in other words to skip to the next line, at the end of its printing. It's one of the elementary formatting instructions that **`printf`** obeys. You'll soon learn how useful it is to be able to do this.

When **`printf`** is called it takes the text as its input and has the effect of writing this text to the screen. It also, separately, comes up with a numerical result, which can be used by a program to determine whether or not the function was successful. This result is ignored by our program, which stops once **`printf`** has done its work.

The final thing of note is the semi-colon at the end of the **`printf`** line. It's another one of those grammatical aspects of C – it marks the end of a statement for the compiler, letting it know that what follows is either another statement or, as in this case, the end of a block, as denoted by a closing curly brace.

And the closing of that curly brace marks the end of your first program. Not a very complex one, granted, but the first step towards great things.

Numbers

One of the more important things that C can do is deal with numbers. Even the most un-numerical of applications will rely heavily on numbers. Although these numbers and their manipulation (usually pretty basic in mathematical terms, don't worry) may be hidden from the program user's view, you as the program's creator must deal with them. Here's how.

Integer

C divides numbers up into several different types, the most common of which is a type called "integer". An integer is a whole number, positive or negative in value. There's a limit to its size, but we'll go into more detail about that later. All the usual mathematical operations can be performed on integers, but if you divide one integer by another you'll end up with an integer result, a whole number. The "remainder" will be forgotten. Dividing three by two will yield a result of one.

Storing numbers – variables

As I said in Chapter One, for numbers to be of any use there must be a means of storing them. For this purpose, and for the purposes of storing other things, C supplies variables. You can think of a variable as the memory location, or more accurately the collection of memory locations, in which the number is held. But rather than using a numerical address, the programmer can access the number stored by means of the variable name, which can be an intelligible word decided on by the programmer.

Variable

Variables are akin to the Xs and Ys of school-day algebra. Numbers may be stored in them and altered. In other words, their contents "vary" throughout the time the program is executing (running).

Why is it necessary to refer to numbers via variables rather than as numbers themselves? Imagine you were writing a spreadsheet program, and that you needed to total up all the numbers in a column. You could define a variable called **total**, and set its value to zero. Then you could add the number in the first column to it. Then the second number would be added, and the third, and so on. After all the additions were complete, the value held in the variable **total** would contain the sum of the column. It would be impossible to write a program to do this if it only referred to numerical values, since these values would be unalterable once the program was running. Nor could any of the numbers in the column be written into the program simply as numbers. If they were, then the spreadsheet would contain exactly the same numbers every time it was run. Their values must be decided by the user when the program is running, in other words they too must be stored as variables, with their values either typed in by the user or loaded in from a file on disk.

To help clarify the above, let's take a look at a simplified program to do the spreadsheet totalling:

COMPLETE LISTING

```
#include <stdio.h>

/* program to total four numbers */
void main()
{
     /* first declare variables */
     int first, second, third, fourth, total;

     /* set the total value to zero */
     total=0;
     /* now assign arbitrary values to the other variables. In a real-
world example these values would be entered by the user running the
program */
     first=23;
     second=-5;
     third=42;
     fourth=1;
     /* now perform the addition to find the total */
     total=total+first;
     total=total+second;
     total=total+third;
     total=total+fourth;
```

```
    printf("The total is ");
    printf("%d\n",total);
}
```

The program begins in the same way as the last, by calling on the services of **stdio.h** (because we'll be needing the **printf** function again) and by defining the function **main**. Once again, **main** returns no result (although it has the effect of printing something to the screen) and requires no inputs, so the word **void** appears before it and the parenthesis after it (which let C know that it is a function) are empty.

Declaring variables & variable 'types'

After the curly brace comes the line which 'declares' our variables. To declare a variable means to let the compiler know what its name is, and what type it is. In C, all variables must be declared before use. They are usually declared at the beginning of a function, before any of its real "meat".

MAKE A NOTE!

Variable names

We are using integers exclusively, represented by the C word **int**, so this word precedes the names of the variables we want to use. The names of variables can be anything you decide, according to certain restrictions. The first character in the variable's name must be a letter, either upper or lowercase (the two are seen as different, so the names **fred** and **FRED** refer to different variables). The following characters be made up of letters, numbers or underscore characters "_". It's conventional to use predominantly lower case letters for variable names. You cannot use words that represent C instructions.

In the above example, all of the integers were declared in one line, with commas separating them. Instead the declaration could have been written as many lines, like so:

CODE SEGMENT

```
int first;
int second;
int third;
int fourth;
int total;
```

The two styles can be mixed and matched as you see fit, but note that you can only declare several variables with a single type specifier (**`int`**, in our case) if they are all of the same type.

The next line after the comment is used to set **`total`**'s value to `0`. In C, the equals sign means "set whatever is to the left of the equals sign to the value of whatever is to the right".

The next few lines assign arbitrary values to the other variables in much the same way. The assignment to the variable called second makes use of the `-` sign; as you might expect this means that the value stored in the variable is a negative one, in this case `-5`. As it says in the preceding comment, for the program to be of any practical use these values would need to be entered by the user while the program was running. Note how the comment can be split across more than one line without ill effect.

Expressions

The next line is where the real work begins. Again, it contains an equals sign, so an assignment is being made to the variable on the left, i.e. **`total`**. The value being assigned is the result of an "expression".

An expression is just like a sum in maths; it relies on "operands" and "operators". In our example the operands (that is to say, the things to be operated on) are variables, or more correctly the numbers held in the variables, and the operator is the plus sign, representing addition. C has many such operators, including ones to carry out all the basic mathematical tasks.

The result of the expression is gained by applying the operator or operators to the objects supplied. We are adding the values held in **`total`** and **`first`** together (`0` and `23` respectively) and getting the result `23`, which is stored in **`total`**, erasing the old value of `0`. Notice how if a variable is being assigned the result of an expression, and that expression itself contains the variable, then the variable's old value is used in computing the expression, and the variable's value is not changed until the expression has been fully evaluated.

The following three lines add each of the next variables in turn to the value held in **total**. When they have all been added, **total** holds the sum of all of them, the value 61.

An expression can contain more than one operator, and more than two operands. All of the above additions could have been compacted into the following line with the same effect:

```
total=total+first+second+third+fourth;
```

CODE SEGMENT

The final two lines are there to print out the result. The first one prints a text message, as in our first example. The second is more interesting.

Passing parameters

For a start, two "parameters", or values to be given to the function, are included between the parentheses. **Printf** is an exception among C functions in that it can accept a variable number of parameters. The first parameter is a piece of text, the last two characters of which mean, as you know, print a newline at the end. The first two characters inform **printf** that it is to expect a second parameter, and that it is to treat it as an integer.

Different letters after the percentage sign are used for different variable types. The two print statements could have been combined into a single one as follows:

```
printf ("The total is %d\n", total);
```

CODE SEGMENT

As you can see the "**%d**" formatting command tells **printf** whereabouts within the printed text to include the value of the integer. If you changed the line to:

```
printf ("You have %d in total",total);
```

CODE SEGMENT

The output produced would be:

```
You have 61 in total
```

The "**%d**" need not appear at the end of the segment of text to be printed.

Type in the above program, compile it and get it running. Once you have done so, try experimenting with the **printf** statement, splitting it across more than one line, varying the position in which **total** is included in the message, and printing other variables too.

Another thing to experiment with is different types of expressions. As well as additions, you can perform subtractions with the **-** operator, multiplications with ***** and divisions with **/**. Try them and see what the results are.

Operator 'precedence'

If you try mixing several different operators on the same line, you might notice some odd results. For one thing, there is the consequence of integer division as mentioned earlier, meaning that the expression **third/first** (i.e. `42/23`) would give the result `1`, rather than the expected `1.826`. To compensate for this, C provides an operator called the "modulus", that will find the remainder. It is written as a percentage sign, and used just like the operators already discussed. For example, the expression **42%23** would give the result `19`, the remainder of `42` divided by `23`.

Modulus

That's division taken care of, but there some more anomalies waiting to be uncovered. What result would you expect from calculating the expression **23-1+42**? There are two possible ways to go about this: you could subtract `1` from `23`, giving `22`, and then add `42` to arrive at the result of `64`; or you could add `1` to `42`, giving `43`, and then subtract this from `23` to arrive at the result of `-20`. To resolve this apparent ambiguity C has the rule that expressions with multiple operators are evaluated left to right, so the correct answer in this case would be `64`.

Using parentheses

You can force a different order of evaluation by using parenthesis. If the above expression were re-written as **23-(1+42)**, then the correct answer would be `-20`. Any operations inside parentheses are carried out before the others. If there is more than one set of "nested" parentheses – i.e. one set enclosed within another, as in the example **`23-(1+(42-7))`** – then the expression within the "deepest" set of parentheses, the one that is most enclosed, is evaluated first.

Here's another tricky one: **`23+1*42`**. Is the answer to this found by adding `1` to `23` and multiplying the total by `42`, or by multiplying `42` by `1` and adding the result to `23`? In fact, the latter is the correct method. Multiplication, division and modulus are said to have a higher "precedence" than addition and subtraction. This simply means that sub-expressions involving them are evaluated first. Once the question of precedence has been dealt with, then the simple left to right rule is followed. So `1+42*23-5` gives the answer `962`.

As before, parentheses can be used to change the order of precedence. As a rule, it's a good idea to use them in complex expressions, that way you can be sure that C is evaluating your expressions in the order you want.

Real numbers and user interaction

There are two modifications that could be made to the previous example which would greatly improve its usefulness. One would be to make it able to handle real numbers, the kind with decimal points and non-whole values; another would be to let a user enter values to be totalled, rather than the program totalling the same four numbers every time it's run.

Floating point variables

The first modification is straightforward. Happily, C provides a variable type, distinct from **`int`**, which can hold non-whole values. It is signified by the keyword **`float`** before the variable's name in the declaration. **`Float`** is short for "floating point", which is a reference to the way such numbers are stored inside the computer's memory. The specifics of this don't concern us; suffice to say that **`float`**s can contain a whole part, a decimal point, and a fractional part. If we wanted to use **`float`**s instead of **`int`**s, the declaring line in our totalling example could be written as follows:

```
float first, second, third, fourth, total;
```

CODE SEGMENT

One thing to note is that if a number is assigned to a **float**, then it should include a decimal point within it, even if it is zero or a whole number. So the initial assignment for **total** becomes:

```
total=0.0
```

CODE SEGMENT

This is to help distinguish between floating point variables and integer ones. DICE won't complain if you assign a 0 to a **float**, but it's better to stick to the convention of assigning a 0.0.

WARNING

Modulus operator and floats

A natural result of using **float**s rather than **int**s is that divisions yield the correct answers. There is a limit to the accuracy of variables, though, so a certain amount of rounding takes place, but **float**s are accurate to a sufficient number of decimal places to make this rounding inconsequential in most circumstances. Using the modulus operator on floats will result in an error, since a non-integer division does not produce a remainder.

To get some input from the user we use another function from **stdio.h**, a close cousin of **printf** called **scanf**. **Scanf** essentially performs the reverse of **printf**: instead of printing the program's output, it takes the program's input. Like **printf**, there are many subtly different ways in which **scanf** can be used. Some of them are quite complex, so we won't go into them at the moment.

We want to use **scanf** to get a user-entered value for subsequent storage in a variable. To do this we must supply **scanf** with two parameters: the first is a format command, telling it to treat the input as a certain type, analogous to the "**%d**" telling **printf** to treat a variable as an integer; the second is the variable in which the result is to be stored.

MAKE A NOTE!

Variables and functions

The format command we will be using is "**%f**", which tells **scanf** that it is dealing with a **float** variable. "**%f**" must also be used with **printf** when we come to print out a **float**. The second parameter, the variable in which the input is to be stored, must have a "**&**" character before its name. There's no space for a full explanation here, but for now suffice to say that ordinarily when a variable is given to a function as a parameter, only the value held by the variable is given, not the variable itself. What this means is that the function may modify this value without affecting the variable – it is only modifying a copy of the variable which is private to itself. For the purposes of **scanf** this is no good – we want the variable itself to be modified. By prefixing its name with "**&**" we let the function know where exactly in memory the variable is stored, meaning that the function can directly alter the number held in memory and thus the value held in the variable. **Scanf** can store its result. Don't worry if you don't really follow this; you'll understand it better once we've dealt with several other, related concepts. For now all you need to know is that integer and floating point variables must be prefixed by "**&**" when they are used as parameters for **scanf**. The line to get a user-entered number into the **float** variable called **first** looks as follows:

```
scanf("%f",&first);
```

CODE SEGMENT

Like **printf**, **scanf** also produces a numerical value, independent of the one assigned to the variable called **first**. This value is the number of items that were input. It could be assigned to an integer variable, possibly to help with error-checking, as follows:

```
was_an_error=scanf("%f",&first);
```

CODE SEGMENT

but for now we'll ignore it.

The modified spreadsheet is shown below. Type it in and try it.

COMPLETE LISTING

```
#include <stdio.h>

/* program to total four non-integer numbers */
void main()
{
    /* first declare variables */
    float first, second, third, fourth, total;

    /* set the total value to zero */
    total=0.0;
    /* Now get the numbers to be added from the user */
    printf("Enter the numbers to be added\n");
    printf("Enter the first number ");
    scanf("%f",&first);
    printf("\nEnter the second ");
    scanf("%f",&second);
    printf("\nEnter the third ");
    scanf("%f",&third);
    printf("\nAnd the fourth ");
    scanf("%f",&fourth);

    /* now perform the addition to find the total */
    total=total+first+second+third+fourth;
    printf("\nThe total is %f\n",total);
}
```

To compile this successfully you'll need to include the maths library at the linking stage – it's needed by both **printf** and **scanf** to handle **float**s successfully. If you're compiling with DICE, and have named the file **spread2**, then using the following line will do the trick:

```
dcc spread2.c -lm -o spread2
```

Notice how I've used the "**\n**" formatting command in the program's **printf**s to keep the screen from getting too cluttered when the program is running. Notice also the use of the "**%f**" command in the final **printf** to let the function know it is dealing with a **float** variable.

Feel free to modify the program as you see fit. In particular, try altering the expression to include multiplication and division. The more you experiment, the more you learn.

Mixing it up

You may be wondering what happens if both **int** and **float** variables are used in the same expression. In general, **int**s are "promoted" to the accuracy of **float**s, so that in an expression involving both, the integer value is treated as a **float** with zero after the decimal point. For example, the result of the expression **`32.3+5`** is `37.3`, not `37`. Similarly, if an integer value is assigned to a **`float`** variable, then it is first converted into a floating point value.

If a floating point value, which may be the result of an expression mixing **`int`**s and **`float`**s, is assigned to an integer variable, then the fractional part is lost.

Decision-making

- Reducing statements to 'true' or 'false'
- 'Switching' statements
- 'If' statements & 'logical expressions'
- Relational operators
- Loops
- 'Goto' statements and 'labels'
- 'Do while...' and 'while...'
- Logical operators (AND, OR, NOT)

If there is one thing about computers that makes them more than glorified adding machines, and one thing that leads to the popular misconception that they can think, it is their ability to make decisions. This ability in no way implies that computers have free will, but is nevertheless so important, so fundamental, that all but the most basic programs rely on it.

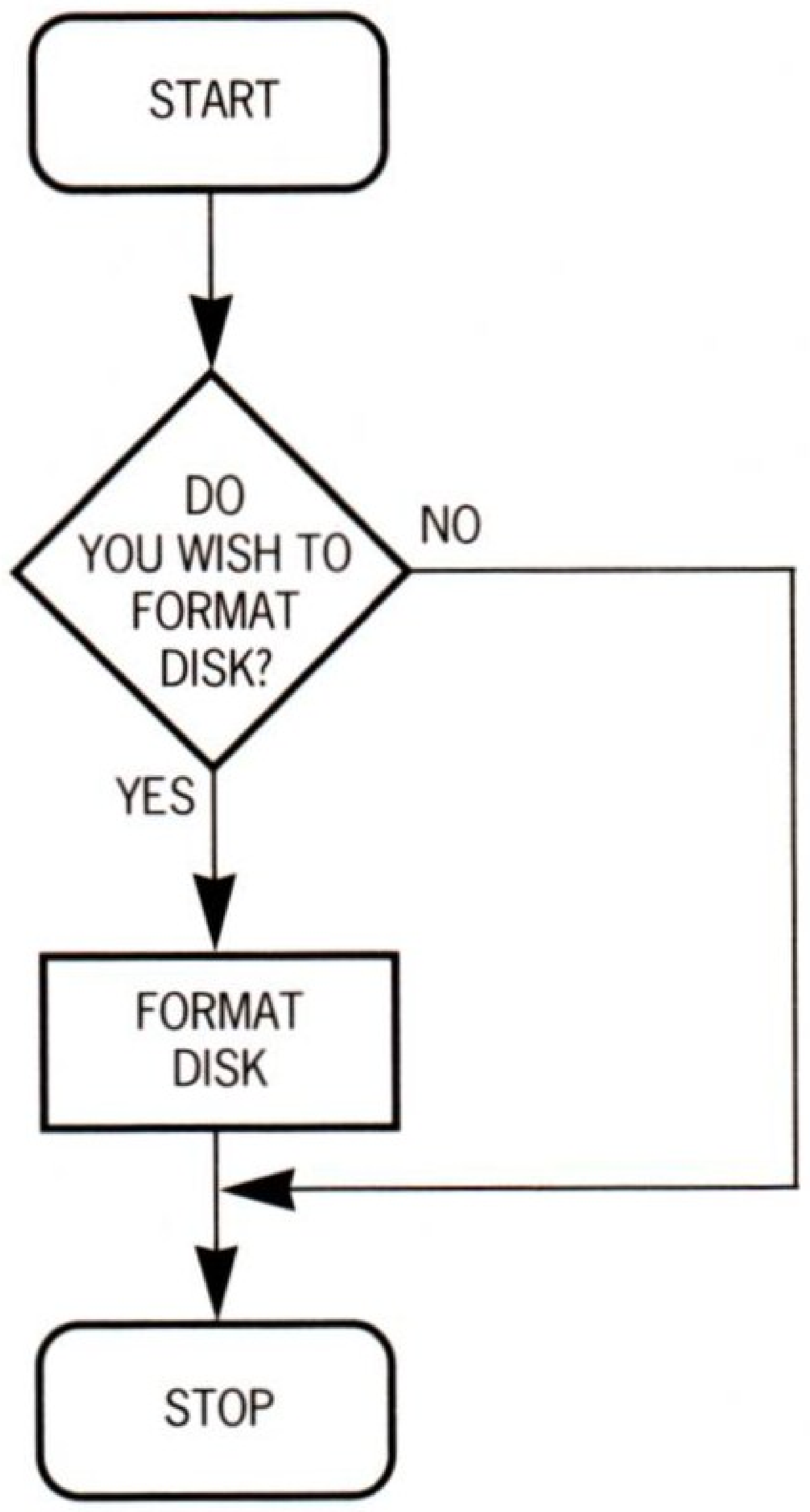
So far we've dealt with programs that follow a single course of action, where each statement within a program is executed in sequence. Decision-making involves us in providing more than one possible course of action. Which of these two courses is taken depends on whether or not a set of circumstances, as defined by the programmer, have been met. Think of a requester that asks a user whether or not the program it belongs to should continue, a good example being a simplified "are you sure? (1 for yes, 2 for no)" requester that appears before a disk is formatted. If the user enters a 1, then one course of action, that of the program going on to format a disk, occurs; if the user enters a 2, then another course is taken, that of the program stopping; if the user enters anything else, then the question is asked again.

The circumstance on which the decision depends is the user's input. This would be taken from the keyboard, perhaps using the **`scanf`** function introduced in the last chapter, and stored in a variable. The decision is then made by looking at the contents of this variable, and seeing if they are the same as any of the expected responses.

In fact, all computer decision-making comes down to the examination of variables, and there is only a small number of ways in which these examinations can be done. Before we look at these, though, let's talk about truth....

True or false?

A statement can be either true or false; the computer has no conception of things being *partially* true. For the example above, we could make the statement, "**`The user entered a 1.`**" It may be true, but if the user entered something else, then it is obviously false. Decision-making involves asking the computer whether or not a particular thing is true. We might, in English, write the decision-



Simple 'Yes/No' decision-making program design

A program can be designed to follow more than one course of action depending on the user's input. In this case, if the user doesn't want to format the disk, the program skips that step.

making part of the program as, "**`Did the user enter 1? If so, format the disk.`**"

What we would actually do is look at the variable – let's assume it's called **`reply`** – in which the user's input was stored. We would compare the contents of the variable with the number `1`, which is a "constant expression" – its value, unlike that of the variable, never changes. We would test to see if the contents of the variable and `1` are one and the same thing, to see if they are equal. The test for equality is the simplest of the possible examinations performable on variables.

Imagine a simple calculator program. It asks the user to enter two numbers, say floating point numbers, and then asks the operation to perform on them: addition, subtraction, multiplication or division. For simplicity's sake, let's make the last input a number, too. We'll use a simple menu system, using '`1`' for addition, '`2`' for subtraction, and so on.

The first part of the program is executed no matter what. It gets two numbers from the user. The second part asks the user for a third number, and then, depending on what that third number is, one of four actions is taken. Finally, the result is printed out to the screen. This last part, like the first, is executed no matter what the user enters.

Here's the program:

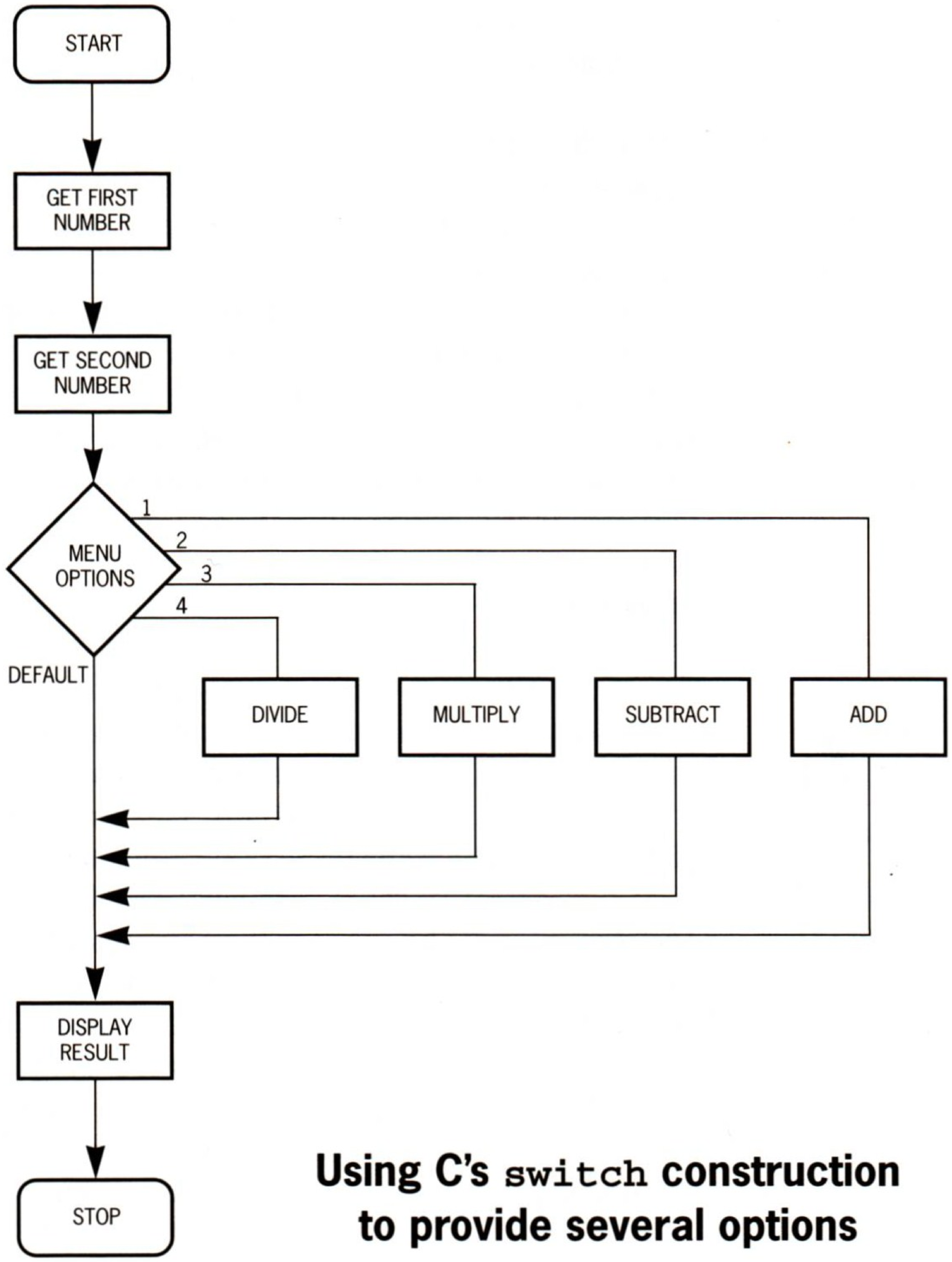
COMPLETE LISTING

```
#include <stdio.h>

/* simple calculator */
void main()
{
     /* declare the variables */
     float first, second, result;
     int reply;

     /* get the user to enter the numbers */
     printf("Enter the two numbers to be operated on \n");
     scanf("%f",&first);
     scanf("%f",&second);
     /* print up the menu and get user's choice */
     printf("Which operation do you require?\n");
     printf("1 - addition\n2 - subtraction\n3 - multiplication\n4 ¬
- division\n");
     scanf("%d",&reply);

     /* now to make the decision */
     switch (reply) {
          case 1: result=first+second;
               break;
          case 2: result=first-second;
               break;
          case 3: result=first*second;
               break;
          case 4: result=first/second;
               break;
          default:
                break;
        }
        printf("\nThe result is %f\n",result);
}
```



Using C's `switch` construction to provide several options

Not all decisions have to be of the simple 'YES/NO' variety.

Everything up to the line which begins with **switch** should be familiar now, but the decision-making part introduces new elements. The **switch** statement means "switch execution to one of the following groups of statements, depending on the result of the expression inside the parentheses." The expression in this case is simply the variable **reply**, but it could be more complex. The body of the switch statement is in curly braces – this means the body constitutes a statement block.

Just in case

Each of the lines beginning with the word "**case**" also represents an expression. If the expression held in the parentheses after the **switch** statement is equal to the expression following **case** (in this "case", ahem, one of four numbers or the word "**default**"), then the statements following **case**'s colon are executed. In our example, the first **case** statement is asking whether or not the variable **reply** holds the value 1. If it does, then the sum of the variables **first** and **second** is stored in the variable **result**, by a statement that ought to be familiar – **result=first+second**. It is followed by a semi-colon, as are all statements that aren't followed by a block enclosed in curly braces.

Break

The next statement – **break** – is an interesting one. It tells the computer to "break out of" the **switch** statement, to ignore all other statements in the block, and continue execution at the next statement not associated with the **switch** statement, in this instance the **printf** that outputs the result of the calculation.

The reason for the **break** is that otherwise execution would continue at the next available statement, which is a check to see whether or not **result** holds the number 2. If the addition statement has been executed we already know that **result** holds the number 1 and we've done everything we want to do that depends on the fact, and that there's therefore no need to perform any further checking, and it's time to get on with the rest of the program.

Default

For the same reason, the **break** statement follows the statements to be performed after each of the checks for the other possibilities (2, 3, 4 and the mysterious "default"). The last one – default – is used to catch any unexpected result. If, in our example, the user had entered anything other than `1`, `2`, `3` or `4`, then execution would end up here. You could put a statement that prints out an error message here, something like, "`You have not entered a valid option.`" Try it.

As it is though, there is no statement following **`default:`**, other than **`break`**. It's still important to put this segment in, though. Although a **`break`** after **`default:`** isn't necessary, it's a good idea to include it. Apart from its inclusion being a convention, it's a good guarantee against your introducing a bug should you decided in the future to add another case to the end of your **`switch`** statement, e.g. an option `5` to find percentages in the above program. (Otherwise your program would find a percentage whether its user entered `5`, `6`, `7` or any other invalid option.)

MAKE A NOTE!

Switch statement

The **`switch`** statement provides a good way of performing one of several actions, depending on the contents of a variable. A common use for it, as above, is in responding to a user's menu selection. There are many more uses for it, however, as you'll discover. An important point to remember is that more than one statement can come between a particular case and the next, so it offers a great deal of flexibility.

ifs but no buts

Sometimes, though, you may want to make simpler decisions, with fewer possible outcomes, or decisions that rely on a variable holding something other than integer numbers (for that is all **`switch`** can decide between). In these situations it's more sensible to use C's **`if`** statement.

With **`if`**, you can ask whether something is true, and if it is, then do something. If it isn't true, then the statement, or series of statements enclosed in curly braces, that comprise the "do something" bit are ignored. Here's a quick example:

COMPLETE LISTING

```
#include <stdio.h>

/* Quick demo of if */
void main()
{
     int reply;
     printf("Enter the code number\n");
     scanf("%d",&reply);
     if (reply==999)
          printf("Correct\n");
}
```

OK, it doesn't do very much, but I've kept it simple to illustrate the point. The bit inside **if**'s parentheses is a "logical expression". If it is true, then the statement immediately following is executed. If not, it's ignored and, in this example, the program finishes.

The expression here is a comparison between two smaller expressions (the contents of the variable **reply** and the number 999). The program is testing to see whether or not they are equal.

In C, the test for equality is done using two equals signs – **==** – together, with an expression on either side. If the results of the two expressions are the same, then the test is said to be true, otherwise false. For example, **2==3** and **7.9==7** are false, while **1==1** and, if the variable **reply** holds the number 1, **reply==1** are true. Two equals signs together are used because one on its own, as you should remember, is used to store the result of an expression (which may be a simple expression, such as a number) in a variable. Confusing = and == is one of the most common mistakes made by C programmers.

The comparison as a whole, made up as it is of two expressions and the test for equality (known as a "relational operator", since it examines the relationship between two expressions), can also be viewed as an expression. Unlike other expressions, which can have a whole range of values, this kind of expression can have only one of two logical values: truth and falsity. Falsity is represented by the number 0, any other number is equivalent to truth. This means that you could have an **if** test without using a relational operator. In other words, you could substitute a simple expression for a logical one involving a relational operator.

It would come in handy for a program dividing two integers. As you may know, there's no sensible result gained from dividing any number by zero. The answer tends towards infinity, and always generates a computer error. So your program would need to check to make sure the divisor was something other than zero. You could write it like this:

COMPLETE LISTING

```
#include <stdio.h>

/* simple division program */
void main()
{
     /* declare variables */
     int dividend, divisor, result;

     /* get their values from user */
     printf("Enter the two numbers to be divided\n");
     scanf("%d",&dividend);
     scanf("%d",&divisor);

     /* test to see if divisor is not zero */

     if (divisor) {
     result=dividend/divisor;
     printf("The answer is %d\n",result);
     }
}
```

You could, if you liked, streamline this a bit as follows:

COMPLETE LISTING

```
#include <stdio.h>

/* simple division program */
void main()
{
    /* declare variables */
    int dividend, divisor;

    /* get their values from user */
    printf("Enter the two numbers to be divided\n");
    scanf("%d",&dividend);
    scanf("%d",&divisor);

    /* test to see if divisor is not zero */

    if (divisor)
    printf("The answer is %d\n",dividend/divisor);
}
```

It needs one less variable, and loses a line of code. **`Printf`** only requires a value to be passed to it, not necessarily a variable. So here we've given it the value found by evaluating an expression. Notice also how the curly braces surrounding the consequences of the **`if`** statement can be dropped if the consequence is only made up of a single statement.

Relational operators

Different kinds of logical expressions can be formed by using different types of relational operators. As well as testing to see whether two things are equal, you can test to see if one is larger than the other, or if one is smaller than the other.

The symbols used for these are as follows:

>	Pronounced "greater than", which tests to see if the expression on the left is numerically bigger than the one on the right
>=	("greater than or equal to") Tests to see if the expression on the left is at least equal to, if not bigger than, the one on the right
<	("less than") tests to see whether the expression on the left is smaller than that on the right

<= (“less than or equal to”) tests to see whether the left-hand expression is smaller than or at most equal to that on the right.

You already know about **==** (“equals”) which tests to see if the two expressions are equal. There’s a similar, but opposite operator, which tests to see if the two expressions are *not* equal. It is written **!=** (and is pronounced, unsurprisingly, “not equal to”).

In the last example, we used the test **`if (divisor)`** to ensure the division is only performed if divisor equals something other than zero. It might be clearer to write the test out more explicitly:

```
if (divisor!=0)
    printf("The result is %d\n",dividend/divisor);
```

The other relational operators (“less than” and the rest) are useful in a whole range of circumstances, but, rather than concoct an artificial example right now, I’ll use them as they become necessary in later programs.

Doing it again (and again)

Let’s go back to the first example of the chapter, the simple calculator. We left it by saying some sort of error message segment would be useful for if the user entered a number other than `1`, `2`, `3` or `4`. If you modified the program yourself, then the last part of your switch statement will look something like this:

```
        default:
            printf("You have not entered a valid option\n");
            break;
    }
```

(If you did do it yourself, I hope you didn’t miss out the semi-colons at the ends of **`printf`** and **`break`**; and if you didn’t do it yourself, why not?) This solution is all very well, but the program will, if the user enters an invalid choice, give him a result of `0` and has to be run again before he can enter the choice he actually wants.

What we need is an opportunity for error recovery, like this:

```
default:
     printf(“You have not entered a valid option ¬
     - try again\n”);
     printf(“1 - addition\n2 - subtraction\n3 ¬
     - multiplication\n4 - division”);
     scanf(“%d”,&reply);
     break;
}
```

Have you seen the problem with this approach? If not, add it to the calculator program and run the whole lot. Although we've given the user the opportunity to correct his mistake and enter a new option, we haven't modified the program so that it will do something *with* this new option.

One, particularly clumsy, solution is to include a copy of the whole of the unmodified **switch** statement after the first **switch** statement's **default**, so the whole decision process could be made again, depending on the new value of reply. It would look something like this:

```
default:
     printf(“You have not entered a valid option ¬
     - try again\n”);
     printf(“1 - addition\n2 - subtraction\n3 ¬
     - multiplication\n4 - division”);
     scanf(“%d”,&reply);
     /* now to make the decision again */
     switch (reply) {
          case 1: result=first+second;
             break;
          case 2: result=first-second;
             break;
          case 3: result=first*second;
             break;
          case 4: result=first/second;
             break;
```

```
                    default:
                        break;
                }
                break;
        }
        printf("\nThe result is %f\n",result);
}
```

Now, this is an example of pretty bad programming. It will work – add it to the original, and see – but not particularly well. If the user enters an incorrect value twice, then he'll still end up without a proper result, and he'll have to run the program again (you might argue that, after two chances, he deserves all he gets, but it's just as easy to write a program that gives him as many chances as it takes).

Secondly, we've had to write out the majority of the program twice. The logical extension of this is that for every chance we give the user to re-think his input we're going to need a corresponding piece of code, even though it is the same as all the others.

It might be a bad solution, but it does at least illustrate one point: that you can embed, or "nest", **case** statements inside other **case** statements. You can also nest **if** statements in this way, so you can test for conditions only if certain other conditions have already been met. This sort of thing comes in useful in more complex programs. For now, back to the error recovery problem.

Jump to it

WHAT DOES IT MEAN?
Goto statement

A slightly better solution is to use a jump. C provides a statement called **goto**, which enables you to transfer execution to anywhere else in the program. The point you want execution to continue from is marked by a label, a piece of text that obeys the same rules as the name of a variable, followed by a colon. The same label also appears immediately after the **goto** statement.

You can use **goto** to jump forwards in the program, skipping intervening statements, or backwards, to execute a segment of code for a second time. It's here that it becomes useful for our calculator program. Take a look at this latest version:

COMPLETE LISTING

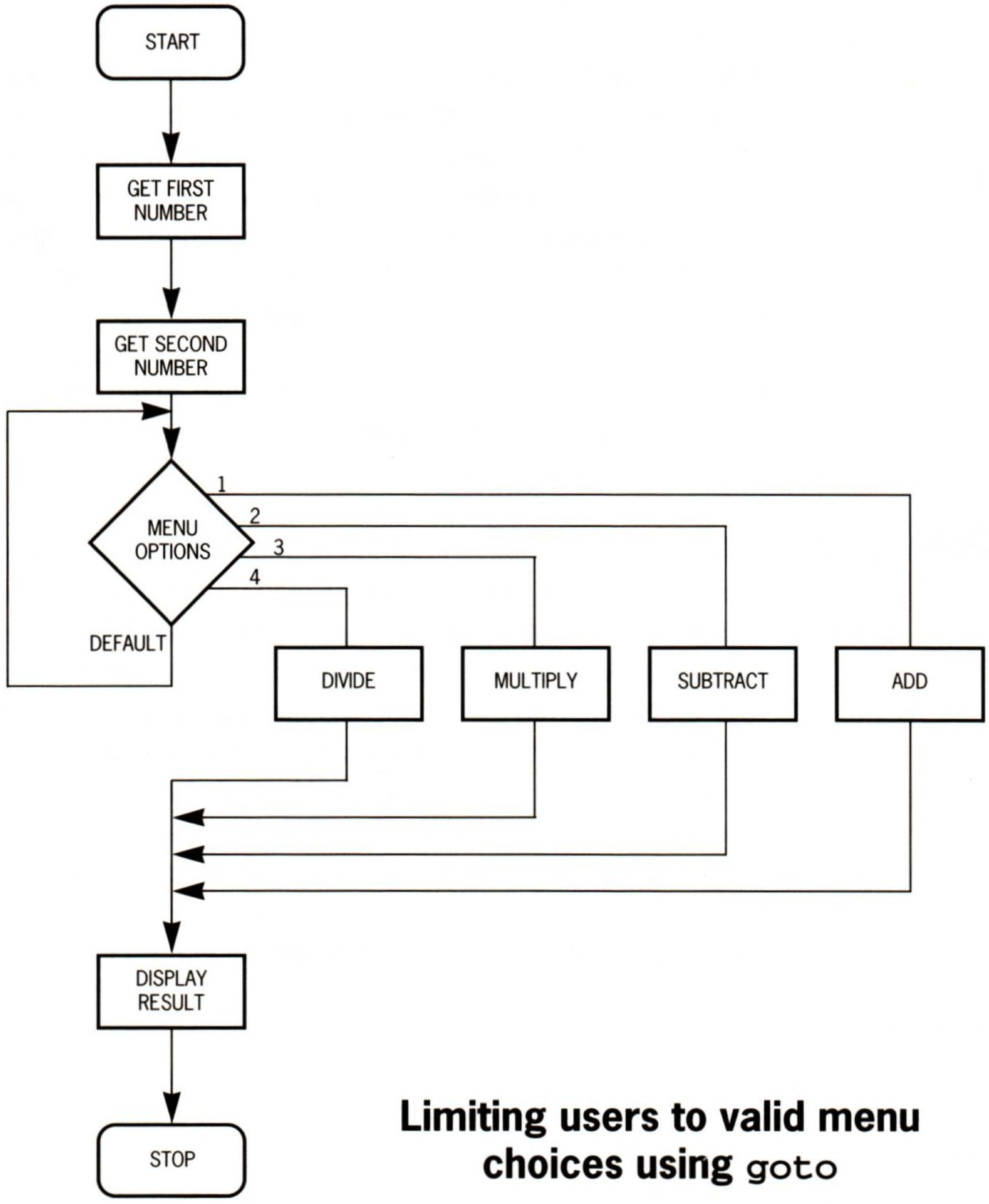
```
#include <stdio.h>

/* simple calculator for careless users */
void main()
{
     /* declare the variables */
     float first, second, result;
     int reply;

     /* get the user to enter the numbers */
     printf("Enter the two numbers to be operated on\n");
     scanf("%f",&first);
     scanf("%f",&second);
     /* print up the menu and get user's choice */
     /* next lines is the label - control jumps back here if the user
enters anything other than 1, 2, 3 or 4 */
     try_again:
     printf("Which operation do you require?\n");
     printf("1 - addition\n2 - subtraction\n3 - multiplication\n4 ¬
- division\n");
     scanf("%d",&reply);

     /* now to make the decision */
     switch (reply) {
     case 1: result=first+second;
          break;
     case 2: result=first-second;
          break;
     case 3: result=first*second;
          break;
     case 4: result=first/second;
          break;
     default:
          printf("You have not entered a valid option - try again\n");
          /* jump to the part of the program that reads the user's
menu selection */
          goto try_again;
          break;
     }
```

```
    printf("\nThe result is %f\n",result);
}
```



Limiting users to valid menu choices using `goto`

The goto command is useful for going straight to a specific point in the program. In this case, if the user doesn't enter one of the menu options he's taken straight back to the menu again. The only way out is to choose a 'valid' menu option.

Now this is much more elegant. We don't even need two segments of code to input the user's choice – the first instance just gets used again. Also, the user can enter as many wrong values as he likes, and the program will continue patiently asking for another until a valid one is input.

The situation is still not ideal, however. **Goto** statements are generally frowned on by programmers, the reason being that you have to look pretty hard at a program using them to see what's going on. In the above example, having seen the **goto** statement you then have to go searching through the listing to find out to where the jump is made. It isn't until you've also examined the **switch** statement that the logic of the loop becomes apparent.

C provides a few ways in which you can make this logic, and the logic of similar repeating segments, explicit.

Doing the do

The one that's of most use here is made up of two statements: **do** and **while**. They surround a statement (or several statements grouped into a block by curly braces) which is executed for as long as the logical expression contained within the parentheses that follows **while** is true. For this loop (as segments of code that are executed more than once are called) to be of any use, the intervening statements must modify the variable used in the expression, otherwise the loop may be repeated indefinitely.

Here's how we would use it with our calculator program:

COMPLETE LISTING

```
#include <stdio.h>

/* simple calculator for careless users */
void main()
{
     /* declare the variables */
     float first, second, result;
     int reply;
     /* get the user to enter the numbers */
     printf("Enter the two numbers to be operated on\n");
     scanf("%f",&first);
     scanf("%f",&second);
     /* print up the menu and get user's choice */
     /* This next statement marks the beginning of the loop, which
will be executed over and over until the user enters a valid menu
selection */
     do {
          printf("Which operation do you require?\n");
          printf("1 - addition\n2 - subtraction\n3 - multiplication\n¬
4 - division\n");
          scanf("%d",&reply);
     } while (reply<1 || reply>4);

     /* got a decent reply - now to make the decision */

     switch (reply) {
          case 1: result=first+second;
               break;
          case 2: result=first-second;
               break;
          case 3: result=first*second;
               break;
          case 4: result=first/second;
               break;
          default:
               /* no need for anything here, since this part of the
program should never be executed */
               break;
     }
     printf("\nThe result is %f\n",result);
}
```

This looks much clearer. From this it should be obvious that the bit in curly braces is executed for as long as the logical expression after the **while** statement is true. Quite what this expression is I'll come to in a moment, after I've pointed out the lack of any statements other than **break** following the **default** clause in the **switch** construct. There's no longer need for any, since we've already checked that result holds one of the four valid numbers. Nevertheless, we have included **`default:`** and **`break`**, by convention.

Logical operators

Now to the expression. It is actually a combination of two, joined together by a logical operator called "**`or`**". The "**`or`**" operator is written as **`||`**. It combines the expression on its left with the one on its right (in the example **`reply<1`** and **`reply>4`**) in such a way that if either of the expressions is true, then the total, overall expression is true. If both are false, then the expression containing them is also false. You might express the logic of the loop in words as, "Carry out the segment within the loop for as long as **`reply`** is less than one or it is greater than four." Only when it is one of the correct values will the loop be left and the rest of the program, the calculation, be executed.

WHAT DOES IT MEAN?

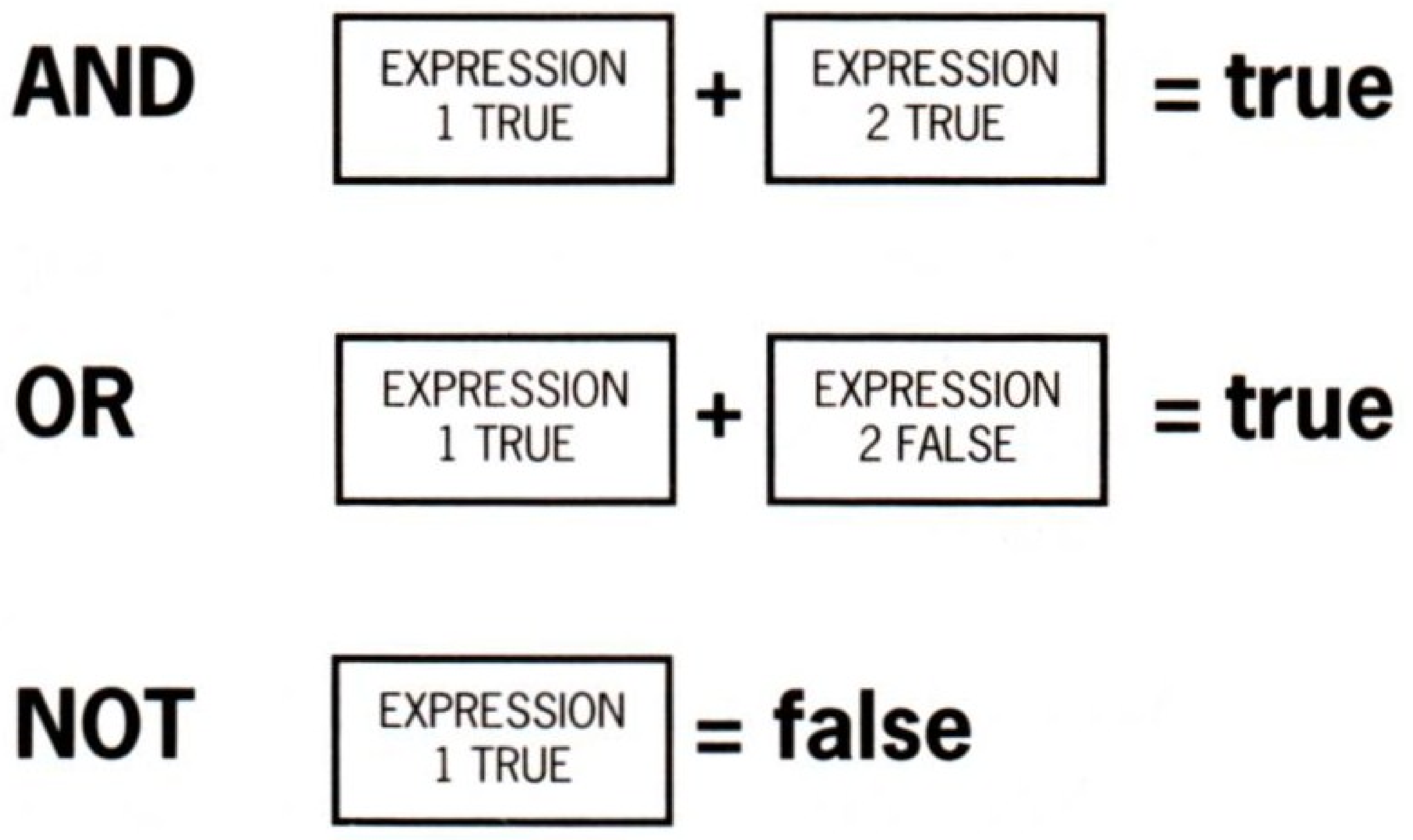
OR locigal operator

Notice that the expression is only tested for its truth after the statements within the loop have been executed at least once. A consequence of this is that, in a longer program, it is conceivable that the variable **`reply`** might already hold a valid value before the loop was entered, yet the program would still ask the user to enter another. You could avoid this situation by placing the expression at the beginning of the loop. Change the loop segment to the following:

CODE SEGMENT

```
while (reply<1 || reply>4) {
     printf("Which operation do you require?\n");
     printf("1 - addition\n2 - subtraction\n3 - ¬
     multiplication\n4 - division");
     scanf("%d",&reply);
}
```

It's similar to the previous loop, but no **`do`** keyword is required. The end of the loop is marked by a closing curly brace. The contents are only executed if the condition after the **`while`** keyword is true. This isn't a particularly good example of the **`while`** loop's use, since the variable **`reply`** doesn't hold anything before it is tested. You would need to set it up with a dummy value beforehand, with a statement



Logical operators

*The logical operator AND returns true if **both** expressions are true. OR returns true if **either** of the expressions is true. While NOT simply returns the **opposite** of an expression.*

such as **`reply=0`**;. Once variables are declared, and before you've put into them, their values are said to be undefined, which means, practically speaking, they can contain any old value. In our example it's possible that **`reply`** could contain one of the four valid values, in which case the program would go off and do a calculation without the user being given a choice as to which one.

DO WHILE and WHILE loops

The difference between the **`do  while`** and the **`while`** loops is whether or not the loop is executed at least once as a matter of course, or whether the test has to prove true before the loop is executed at all. The **`do  while`** loop ensures the former, the **`while`** loop the latter. In our example, we want the loop always to be executed at least once, in other words for the user to have at least one chance to enter an option, so the best of the two is the **`do  while`** loop. There are circumstances in which the **`while`** loop is more convenient, but we'll come on to these in due course.

AND logical operator

There's another logical operator, similar to the "**`or`**" used above, called "**`and`**". It is written as **`&&`**. Logical expressions using "**`and`**" are only true if both the expression on the left and the expression on the right of the **`&&`** sign are true. If either or both are false, then the overall expression is also false.

The final logical operator, "**not**", is different in that, rather than combining two smaller logical expressions, it operates on only one. The result of an expression involving **not** is the logical inverse of the expression following the symbol. It turns a true expression into a false one, and vice versa. It is written as **!**, followed by a logical expression.

We could combine the "**and**" and "**not**" operators to come up with a different approach to the loop logic in the above example. Whereas the logic of the **do while** loop was expressed as, "Carry out the segment within the loop for as long as reply is less than one or it is greater than four," we could write a loop with the logic, "Carry out the segment within the loop for as long as reply is not greater than or equal to one and less than or equal to four." The C equivalent would look as follows:

CODE SEGMENT

```
do {
     printf("Which operation do you require?\n");
     printf("1 - addition\n2 - subtraction\n3 - ¬
     multiplication\n4 - division");
     scanf("%d",&reply);
} while (!(reply>=1 && reply<=4));
```

Let's look at the expression within the deepest set of parentheses. It is true only if **reply** is greater than or equal to 1 *and* **reply** is less than or equal to 4. It is enclosed in parentheses, and its result is converted to its opposite by the "**not**" operator that precedes it. The loop therefore executes so long as the result of the inner logical expression (which is checking for a correct user-entered value) is false, that is to say so long as the user has entered an incorrect value.

You'll find that "**not**", "**and**" and "**or**" are used a lot, not just in **while** and **do while** loops, but also in **if** statements and a new statement to be introduced next chapter. They enable more complex decisions to be made by a single statement, that is to say they enable different actions to be performed based on more subtly distinguished criteria. You'll find yourself using them more and more as you create more involved programs.

THE SMALL PRINT

C is the most popular professional programming language in use today. Much of the Amiga's operating system was written in C, and learning this language won't just let you produce professional Amiga applications, it will give you a grounding in C programming on any platform.

This book is supplied with the DICE C programming environment. This is acknowledged to be one of the best versions of C available. The copy supplied with this book is the full, registered version, complete with floating-point maths, on-disk documentation and all necessary Commodore libraries and includes.

Complete Amiga C is written by Cliff Ramshaw, ex professional programmer and now editor of Amiga Shopper.

AMIGA SHOPPER

Complete **Amiga C**

This book is just a sample of the real thing. "Complete Amiga C' is designed as a complete, self-contained C programming environment providing everything you need to start coding:

- **The book itself** 300+ pages covering programming basics right up to advanced concepts and semi-professional programming
- **DICE** Not the 'freeware' version in the public domain, but the fully-featured registered version
- **On-disk reference** As supplied with the registered version of DICE
- **Commodore includes** Essential files for fully releasing your Amiga's graphics & sound abilities

A commercial C package could cost you £200. 'Complete Amiga C' costs just £24.95!

Future PUBLISHING

Your guarantee of value

'Complete Amiga C' is available in all good bookstores. You can also order direct from Future Publishing. See Amiga Shopper for more details.

'Complete Amiga C' – a complete C programming environment – if you like this sample, get the book!